Automatically Identifying Compiler Performance Anomalies

Tipp Moseley
Dirk Grunwald
University of Colorado at Boulder

Ramesh Peri
Intel
Primitive Development Cycle for Compilers

- Come up with an idea
- Test it
- If nothing improved, go to 1
- Celebrate
Understanding transformations is hard

• Code size vs. control flow
  – Inlining, loop transformations, superblocks, if-conversion

• Architectural Features
  – Superscalar, OoO, speculation, EPIC, multilevel memory hierarchy, prefetching
Unpredictable Results

445.gobmk Jump Instruction Comparison

Function
- countlib
- remove_liberty
- verify_stored_board
- assimilate_string
- update_liberties
- propose_edge_moves
- chainlinks2
- hashtable_clear
- incremental_order_moves
- fastlib

Normalized Count
- intel 9.1
- gcc 4.1.0
Unpredictable Interactions

• Dozens of optimizations and parameters
• Selecting per-bench parameters for gcc on SPEC improves performance by 6%
• Best overall loop unrolling factor for 132.ijpeg is 2, but performance increases 8.81% if each function uses best parameter
• Iterative Compilation tries many combinations
  – Balances between compilation speed and search depth
Outline

• Relative Profile Data Analysis (RPDM)
  – Methodology
  – Screenshots
• Case Studies
• General Observations
• Future Work
Relative Profile Data Analysis

• Provide detailed metrics to measure impact of differently compiled benchmarks
  – Function-level comparison

• Usage scenarios
  – Identify missed opportunities and performance bugs
  – Understand impact of new optimizations
  – Regression testing
Relative Profile Data Analysis

1. Collect instruction mix profiles for many benchmarks with multiple compilers and optimization flags
2. Populate database with profile data
3. Brute force query database to identify most significant outlier functions (e.g., total ins, FP ops, jumps, stack r/w, mem r/w)
4. Visually inspect interesting cases
Step 1: Select some profiles (you probably want to select profiles for the same architecture you're using.

<table>
<thead>
<tr>
<th></th>
<th>linux em64t gnu 4.1.0 shared -02</th>
<th>linux em64t gnu 4.1.0 shared -03-mtune</th>
<th>linux em64t intel 9.1 shared -02</th>
<th>linux em64t intel 9.1 shared -03-mtune</th>
<th>linux ia32 gnu 3.2.3 shared -fast</th>
<th>linux ia32 gnu 3.4.6 shared -02</th>
<th>linux ia32 gnu 4.1.0 shared -02</th>
<th>linux ia32 gnu 4.1.0 shared -03-mtune</th>
<th>linux ia32 gnu 4.1.1 shared -02 ref0 -02</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu2000/253.perlbmk/train/0</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cpu2000/253.perlbmk/train/1</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cpu2000/253.perlbmk/train/2</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cpu2000/253.perlbmk/train/3</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cpu2000/253.perlbmk/train/4</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cpu2000/253.perlbmk/train/5</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cpu2000/253.perlbmk/train/6</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cpu2000/253.perlbmk/train/7</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cpu2000/253.perlbmk/train/8</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Current Query: '^J'**

**Matched Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>MAX</th>
<th>normalized</th>
<th>normalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{dynamic-counts}$</td>
<td>1B</td>
<td>1.000</td>
<td>0.423</td>
</tr>
<tr>
<td>$\text{scalar_mult_add_su3_matrix}$</td>
<td>404M</td>
<td>1.000</td>
<td>0.333</td>
</tr>
<tr>
<td>$\text{su3_projector}$</td>
<td>254M</td>
<td>1.000</td>
<td>0.333</td>
</tr>
<tr>
<td>$\text{mult_su3_nn}$</td>
<td>206M</td>
<td>1.000</td>
<td>0.333</td>
</tr>
<tr>
<td>$\text{mult_su3_na}$</td>
<td>201M</td>
<td>1.000</td>
<td>0.333</td>
</tr>
<tr>
<td>$\text{su3_adjoint}$</td>
<td>74M</td>
<td>1.000</td>
<td>0.333</td>
</tr>
<tr>
<td>$\text{mult_su3_mat_vec}$</td>
<td>63M</td>
<td>1.000</td>
<td>0.000</td>
</tr>
<tr>
<td>$\text{mult_adj_su3_mat_vec}$</td>
<td>61M</td>
<td>1.000</td>
<td>0.000</td>
</tr>
<tr>
<td>$\text{start_gather_from_temp}$</td>
<td>59M</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>$\text{scalar_mult_add_su3_vector}$</td>
<td>39M</td>
<td>1.000</td>
<td>0.000</td>
</tr>
<tr>
<td>$\text{su3mat_copy}$</td>
<td>28M</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>Category</td>
<td>MAX</td>
<td>Disassemble</td>
<td>Disassemble (dot)</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----</td>
<td>-------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>Dyntotal</td>
<td>5B</td>
<td>100.0</td>
<td></td>
</tr>
<tr>
<td>Stack Reads</td>
<td>437M</td>
<td>100.0</td>
<td></td>
</tr>
<tr>
<td>Stack Writes</td>
<td>134M</td>
<td>100.0</td>
<td></td>
</tr>
<tr>
<td>Memory Reads</td>
<td>1B</td>
<td>100.0</td>
<td></td>
</tr>
<tr>
<td>Memory Writes</td>
<td>740M</td>
<td>100.0</td>
<td></td>
</tr>
<tr>
<td>mem-read-var</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>mem-write-var</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>mem-atomic</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>iprel-read</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>iprel-write</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>CMOV*</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>ADD*</td>
<td>1B</td>
<td>100.0</td>
<td></td>
</tr>
<tr>
<td>SUB*</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>J*</td>
<td>404M</td>
<td>100.0</td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>RET</td>
<td>33M</td>
<td>100.0</td>
<td></td>
</tr>
<tr>
<td>MUL*</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>DIV*</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>
register int i,j;
for(i=0;i<3;i++) {
    for(j=0;j<3;j++) {
        c->e[i][j].real =
        a->e[i][j].real + s*b - e[i][j].real;
        c->e[i][j].imag =
        a->e[i][j].imag + s*b - e[i][j].imag;
    }
}

Case Study: 445.gobmk countlib()

Intel

GCC: 800% Faster!
/* Count the number of liberties of the string at pos. pos must not be empty. */

int countlib(int str)
{
    ASSERT1(((board[str]) == WHITE || (board[str]) == BLACK), str);

    /* We already know the number of liberties. Just look it up. */
    return string[string_number[str]].liberties;
}
Case Study: 445.namd Patch::zeroforces()

• Both loads are loop-invariant (%ecx does not change)
• GCC 4.1 hoists the loads above the loop resulting in 6912 loads versus 1.6M from ICC
• GCC’s code is 25% faster

```
head:
mov 0x74(%ecx),%esi
fstl (%eax,%esi,1)
fstl 0x8(%eax,%esi,1)
fstl 0x10(%eax,%esi,1)
mov 0x78(%ecx),%esi
fstl (%eax,%esi,1)
fstl 0x8(%eax,%esi,1)
fstl 0x10(%eax,%esi,1)
add $0x18,%eax
add $0x1,%edx
cmp (%ecx),%edx
jl head
```
Case Study: 462.libquantum

- For a simple, linear for loop, ICC inserts a speculative load that often fails (still don’t fully understand why)

- Example of over-aggressive optimization

- GCC’s version executes 60% less instructions and 46% less memory reads, 32.5% faster

IA-64 GCC 4.1.0 –O2

```c
// quantum_addscratch():
for(i=0; i<reg->size; i++) {
  l = reg->node[i].state<<bits;
  reg->node[i].state = l;
}
```

**HEAD:**

```c
{ld8 r14=[r33];;
nop.m 0x0
shl r14=r14,r32;;}
{st8 [r33]=r14,16
nop.i 0x0
br.cloop.sptk.few HEAD;;}
```
Case Study: 464.h264ref
SetCoeffAndReconstruction8x8()

• Innermost loop of quadruply nested loop

• GCC 4.1.0 recomputes address calculations each iteration

• 93% less stack writes, 170% faster

GCC 3.4.6 -O2

head:
mov (%edx,%ebx,4),%eax
mov %eax,(%ecx,%ebx,4)
inc %ebx
cmp $0x40,%ebx
jle head

GCC 4.1.0 -O2

head:
mov 0xfffffffa(%ebp),%edx
mov (%edi,%edx,1),%eax
mov 0xffffff9c(%ebp),%edx
mov (%eax,%esi,1),%eax
mov (%eax,%ebx,1),%eax
mov %eax,0xfffffffb0(%ebp)
mov (%edi,%edx,1),%eax
mov 0xfffffffb0(%ebp),%edx
mov (%eax,%esi,1),%eax
mov (%eax,%ebx,1),%eax
mov (%eax,%ecx,1),%eax
mov %eax,(%edx,%ecx,1)
add $0x4,%ecx
cmp $0x40,%ebx
jle head
Observations

• Finding performance bugs is easy!
  – “Big” anomalies take 1-2 hours of analysis
  – Smaller differences more abundant

• Compilers *still* do silly things
  – Direct jump to next instruction
  – Jump to return instruction
  – Spill all registers, then immediately refill
Future Work

• Collect much more detailed profiles
  – Hardware perf data
  – Load-use distance
  – Stack/heap/text memory references
• Use binary matching to correlate profiles at finer grain (loop, bbl)
• Automate regression testing
Questions?